(2)

Software Technology for Adaptable, Reliable Systems (STARS)

Submitted to:
Electronic Systems Division:
Air Force Systems Command, USAF:
Hanscom AFB, MA 01731-5000:

Contract No:
F19628-88-D-0028

CDRL 1000
BR24 Final Report

July 1, 1990

The Boeing Company
Space and Defense Group
Boeing Aerospace and Electronics
P.O. Box 3999
Seattle, Washington 98124

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	01-JUL-90	

4. TITLE AND SUBTITLE

BR24 Final Report

5. FUNDING NUMBERS

C: F19628-88-D-0028

TA: BR-24

6. AUTHOR(S)

David H. Jones

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The Boeing Company
Boeing Aerospace and Electronics Division
Systems and Software Engineering
P.O. Box 3999
Seattle, Washington 98124

8. PERFORMING ORGANIZATION REPORT NUMBER

D-613- 11000

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

ESD/AVS
Bldg. 17-04
Room 113
Hanscom Air Force Base, 01731-5000

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release - distribution unlimited.

12b. DISTRIBUTION CODE

A

13. ABSTRACT (Maximum 200 words)

Several people in the STARS program working in the area of user interface technology have indirectly contributed to the work presented here: Mark Nelson of SAIC and Kurt Wallnau of Unisys. Bob Rosen of Boeing was the principle contributor to the implementation scheme used in the prototype. The Boeing Commercial Airplane Group's Avionics Flight Systems Central Software also contributed the Ada binding to Motif that was developed by their group.

14. SUBJECT TERMS

Keywords: STARS
virtual interface
X Windows System
user interface
user interaction tasks

15. NUMBER OF PAGES

44

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	None

Name of CDRL

Prepared by

David H. Jones
Chief Programmer (BR24)

Reviewed by

7/2/90

James E. King
System Architect

Reviewed by

7/2/90

John M. Neorr
Development Manager

Approved by

William M. Hodges
STARS Program Manager

Accession For	
NTIS GRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

ABSTRACT

This document reports the results of the Boeing STARS task BR24, User Meta Interface, (henceforth called the "STARS User Interface Toolkit" or SUITE). It includes the status of the prototype work done on BR24 and a comparison code size based on the prototype sample application. There is a technical discussion of the implementation of SUITE; recommendations for future work are identified.

KEY WORDS

user interface
virtual interface
user interaction tasks
X Windows System

PREFACE

Several people in the STARS program working in the area of user interface technology have indirectly contributed to the work presented here: Mark Nelson of SAIC and Kurt Wallnau of Unisys. Bob Rosen of Boeing was the principle contributor to the implementation scheme used in the prototype. The Boeing Commercial Airplane Group's Avionics Flight Systems Central Software (Brian Pflug and Joe Scheer) contributed the Ada binding to Motif that was developed by their group.

KEYWORD STRINGS

user interface
virtual interface
user interaction tasks
X Windows System

TABLE OF CONTENTS

TABLE OF CONTENTS [continued]

LIST OF FIGURES

LIST OF TABLES

1. SCOPE

This document summarizes and reports on the work carried out under the Boeing STARS task BR24, User Meta Interface (henceforth called the STARS User Interface Toolkit (SUITE)). The contents include BR24 results, a technical discussion of the SUITE implementation, and recommendations for future work.

2. REFERENCED DOCUMENTS

This section lists all documents referenced in this Final Report.

[BER89] Berard, E. *Creating Reusable Ada Software,* EVB Training Course, 1989

[BOO87] Booch, G. *Software Components With Ada,* Benjamin-Cummings, 1987

[CDRL980] *Programmer's Guide for STARS User Interface Toolkit (SUITE).* Jan 31, 1990. Boeing STARS CDRL # 980, Electronic Systems Division, Hanscom AFB.

[CDRL970] Ada Code for Meta Level Interface, July 1, 1990. Boeing STARS CDRL # 970, Electronic Systems Division, Hanscom AFB.

[CDRL990] Ada Code for Prototype Sample Application, July 1, 1990. Boeing STARS CDRL # 990, Electronic Systems Division, Hanscom AFB.

[Foley 84] Foley, J.D., et. al.. *"The Human Factors of Computer Graphics Interaction Techniques",* IEEE Computer Graphics and Applications 4(11):13-48, Nov. 1984.

[PER87] Perez-Perez, E. *Simulating Inheritance with Ada,* Ada Letters, Sept. 1987

[WAL90] Wallnau, K. *Ada/Xt Architecture: Design Report,* January 1990. STARS CDRL #01000, Electronic Systems Division, Hanscom AFB.

3. NOTES

This section contains information only and is not contractually binding.

3.1 Abbreviations and Acronyms

APFAT	Ada Program Flow Analysis Tool
API	Application Programmer's Interface
SUITE	STARS User Interface Toolkit
X	X Windows System

4. BR24 RESULTS

4.1 Description of the Prototype Sample Application

The sample application selected for prototyping was the Ada Program Flow Analysis Tool (APFAT), developed by Boeing in the Q phase of the current STARS contract. APFAT was chosen to demonstrate the SUITE virtual interface because it has a variety of static analysis features which produce information that can be displayed in a variety of textual, semi-graphic modes; it can be run in either batch or interactive mode and has a moderate degree of complexity.

In preparation for prototyping, APFAT version 5 was analyzed to determine what modifications would be necessary. In addition, the unreleased document "Software User's Manual for the Ada Program Flow Analysis Tool" was reviewed, which documents an object-oriented user interface that differs from the interface of APFAT V5. Analysis of the two interfaces reveals that the interface described in the "Software User's Manual" is much closer to the style of interface supported by SUITE, in particular:

- It has an object – action selection model.
- The structure is more adapted to a user-controlled dialog.
- The command structure reflects a richer and more highly interactive functionality (but a functionality as yet not implemented in APFAT V5).

As a result of this analysis, we decided to base the user interface on the "Software User's Manual for the Ada Program Flow Analysis Tool (APFAT)." An initial, brief sketch of the new user interface was written and is summarized below:

The APFAT user interface will allow the user to identify 1) particular Ada program objects, 2) information about that object that will be extracted from the symbol table, and 3) display parameters. APFAT operates on objects stored in its symbol table. The types of objects that will be analyzed/displayed include: Identifiers, Task, Subprogram, Package , File, Exception, Interface, and File. The main application panel has standard file and help menus, plus menus to select the Ada entities, identifier, and attributes that should be extracted from the symbol table and displayed. The Ada entity to be analyzed is selected from a single choice list. The Ada program objects (identifiers) for which information is to be extracted from the symbol table are selected from a multi-choice text selection list. A multi choice list of toggle items is used to select the information to be displayed about each selected identifier. The third selection list (not implemented) is a multi-choice list allowing the user to display information on declared objects. The "apply" button causes information about the selected object to be extracted from the symbol table and the output to the viewport.

4.2 Status of Work

In order to demonstrate the feasibility of the concept embodied by SUITE, a prototype sample application was done. To support this prototype, portions of SUITE were also implemented. Only those portions of SUITE that were expected to be directly used by the sample application were actually implemented. As it turned out, this represented a large percentage of the predefined SUITE components: The only components not used by APFAT are valuators (SUITE components that permit the user to view/enter a value within a range of values), several of the dialog panels, text components, viewport, and command line interface panel.

4.2.1 Implementation of SUITE Components

The implementation of SUITE is a hierarchy of derived types and a number of support packages. The base types, from which all other types are derived are completely implemented, with the exception of resource management. (Resource management implements user preferences in an implementation independent way.)

The following components are implemented to a full enough extent to support the prototype sample application: Command Items, Toggle Items, Menu Command Items, Command Lists, Multi-Choice Lists, Application Panels, Applications, File Selection Dialogs.

The following components are NOT implemented to a full enough extent to support the prototype sample application:

- Dialog Panels – Not tested with Frames inserted in them.
- Frames – Cannot insert components in the frame.
- Text Selection - This component was not part of the original specification. It is needed to support the selection of Ada identifiers for which entries exist in APFAT's symbol table. See section 6 of the present document for a more complete discussion of the "Selection Interaction Task".
- Text Display - This was also a component that was not part of the original specification. Its purpose is to support the porting of existing applications that use Text_IO by minimizing the number of code changes necessary to use SUITE. The text display component simulates portions of the standard package Text_IO, causing output to be displayed in specific SUITE components.

4.2.2 Implementation of the New APFAT User Interface

About 80% of the code necessary to set up the use interface has been written and tested. One submenu has not been generated because its component has not yet been implemented.

There are two application processing routines that pass information between APFAT and the user interface. Both routines are partially implemented.

4.2.3 Modifications to APFAT

Modifications to existing APFAT code turned out to be minimal: One AFPAT routine was modified and one was added. The new routine, Symbol_Display processes a representative set of information from the symbol table, but not all information. All information could be displayed with simple additions to a "case" statement.

4.2.4 Demonstration of APFAT with the New User Interface

A complete demonstration of APFAT with its new user interface is not possible at this writing. The application executes and displays menus, submenus. It supports the selection of files for parsing and the selection of the Ada entity class and the symbol table information to be displayed. The menu for selecting identifiers is not complete; symbol table information is not currently displayed on the screen because the text display component is not completed. Instead output is directed to a file. In spite of these limitations the demonstration gives a very good feel for how APFAT would be used through its new user interface.

In addition, demonstration of the functionality of SUITE components is possible through the substantial test code that was written. The following test routines may be used for this purpose: test.a, test_application_panel.a, test_command_item.a, test_command_list.a, test_core.a, test_dialog_panel.a, test_file_selection.a, test_menu_command_item.a, test_resourced_object.,a, test_selection_item.a, test_selection_list.a, test_text_selection.a, test_toggle_item.a test_widget_operations.a, test_x.a .

4.3 Comparison of Code Size

This section compares code size and functionality of the prototype sample application, APFAT, with the baseline version of APFAT, V5.

4.3.1 Metrics for APFAT, Version 5

Compilation Unit Identifier	File Specification
.....	.....

Ada_Parser	ada_parser_.a
Ada_Parser	ada_parser.a
Ada_Scanner	ada_scanner_.a
Ada_Scanner	ada_scanner.a
Adaptation_Data	adaptation_.a
Adaptation_Data	adaptation.a
Apfat	apfat.a
Lexical_Analyzer	lexical_.a
Lexical_Analyzer	lexical.a
Parse_Compilation_U	parse_cu.a
Report_Generator	report_.a
Report_Generator	report.a
Symbol_Definitions	symbol_def_.a
Symbol_Manipulations	symbol_man_.a
Symbol_Manipulations	symbol_man.a
Symbol_UI	symbol_ui_.a
Symbol_UI	symbol_ui.a
User_Interface	user_interface_.a
User_Interface	user_interface.a

The original APFAT code (V5) used text_io for all input/output and had a command line style interface. The size of the source code modules were as follows:

File: //node_17113/local_user/r24/APFAT/adaptation_.a
17 statements 106 lines

File: //node_17113/local_user/r24/APFAT/report_.a
9 statements 99 lines

File: //node_17113/local_user/r24/APFAT/lexical_.a
12 statements 102 lines

File: //node_17113/local_user/r24/APFAT/ada_scanner_.a
14 statements 151 lines

File: //node_17113/local_user/r24/APFAT/symbol_def_.a
94 statements 256 lines

File: //node_17113/local_user/r24/APFAT/symbol_man_.a
15 statements 109 lines

File: //node_17113/local_user/r24/APFAT/symbol_ui_.a
15 statements 134 lines

File: //node_17113/local_user/r24/APFAT/ada_parser_.a
2 statements 65 lines

File: //node_17113/local_user/r24/APFAT/user_interface_.a
8 statements 81 lines

File: //node_17113/local_user/r24/APFAT/apfat_vdp_.a
1 statements 204 lines

File: //node_17113/local_user/r24/APFAT/adaptation.a
29 statements 194 lines

File: //node_17113/local_user/r24/APFAT/report.a
57 statements 230 lines

File: //node_17113/local_user/r24/APFAT/lexical.a
186 statements 669 lines

File: //node_17113/local_user/r24/APFAT/ada_scanner.a
79 statements 439 lines

File: //node_17113/local_user/r24/APFAT/symbol_man.a
442 statements 1393 lines

File: //node_17113/local_user/r24/APFAT/symbol_ui.a
516 statements 1175 lines

File: //node_17113/local_user/r24/APFAT/ada_parser.a
74 statements 282 lines

File: //node_17113/local_user/r24/APFAT/user_interface.a
110 statements 285 lines

File: //node_17113/local_user/r24/APFAT/parse_cu.a
321 statements 686 lines

File: //node_17113/local_user/r24/APFAT/apfat.a
10 statements 90 lines

Totals:
2009 statements 6750 lines

Executable (Apollo DN3500 (68020, 68881), SR10.1, DomainAda 3.0):
333455 bytes

The user interface portion of APFAT V5 is actually in several packages, but only the following packages were modified in the adaptation for SUITE:

- user_interface.a (file deleted; replaced by suite_ui.a)
- ada parser.a (Text_IO calls eliminated)

4.3.2 Metrics for APFAT with SUITE Interface

As mentioned above, only two of the original APFAT packages were modified or removed. The following packages were added to adapt APFAT to SUITE:

- suite_ui – creates the SUITE user interface for APFAT
- symbol_display – parameterized output routines that extract information in the APFAT symbol table and format it for output as ASCII text.
- parse_file, select_symbol_info – These routines do application processing as a result of user actions such as selection. In general these routines respond to user input that requires application processing.

The user interface portion of APFAT interfaces to the original code of APFAT through only three routines:

- Ada_Parser.Parse_Ada_Source_Files – Takes Ada source file name and parses syntactically correct Ada, building from it a symbol table.
- Symbol_Display.Display_Identifier – Produces a formatted ASCII text of symbol table information for the specified Ada identifier and Ada entity. The kinds of symbol table information that can be displayed include : Declares, Invoked_By, Calls, Arguments, Handler, Raises, References, Raised_In, Body, Used, Visible, With_Unit.
- Symbol_Display.Display_Identifiers_By_Class – Produces a list of identifiers for the given class of Ada entities. This information is used to construct Text_Selection components that display Ada identifiers.

The following table summarizes the metrics associated with all new or modified packages (Numbers represent the Ada statement count):

Package	File	V5	Deleted	New
user_interface	user_interface.a	118	118	0
ada_parser	ada_parser.a	74	10	5
symbol_display	symbol_display.a	0	0	137

suite_ui	suite_ui.a	0	0	151
parse_file	parse_file.a	0	0	41
select_symbol_info	select_symbol_info.a	0	0	37
TOTALS:		192	128	371

Executable (Apollo DN3500 (68020, 68881), SR10.1, DomainAda 3.0):
1762937 bytes

Notes: Count in number of statements. Counts refer to combined count of statements in specification and body. New statement count includes both new and modified statements. The package "symbol_display" represents new functionality that APFAT V5 did not have. At the time of this writing the SUITE user interface to APFAT was not completed; we expect the final count to be between 250 and 300 statements. As can be seen by a comparison of the size of executable image, the use of SUITE and other underlying user interface libraries increase the size by a factor of about 5. The major contributor to the size increase is the size of the underlying user interface libraries, in this case Motif and X Windows.

The version of APFAT that uses SUITE is a significantly richer user interface and is much easier to use. We also consider it very significant that the SUITE version of APFAT permits modification and adaptation of the user interface via "User Preferences." APFAT V5 has no similar capabilities. Given these facts, we consider it impressive that, from the application programmer's point of view, the increase in the number of statements is only about 250, or about 12% of the total number of statements in APFAT.

However, the statement count by itself somewhat underestimates the effort required by the application programmer to use the SUITE interface. The Ada statements used to set-up and run the SUITE interface are somewhat longer and more complicated than would be required of only Text_IO were used.

In evaluating the use of SUITE, several other comparisons would have been interesting, but time did not permit them being made:

a. Compare the number of additional statements that would be required to use another user interface toolkit such as Xt/Athena, Motif, or Presentation Manager.

b. Compare the number of language statement required to use a dialog/presentation language such as Motif/UIL or Serpent/SLANG.

Therefore, our preliminary conclusion regarding source code size are that the use of a high level user interface toolkit such as SUITE represents only a 10-20% increase in

code size over a user interface that is written using Text_IO. The significant advantages of using an higher level toolkit such as SUITE are: User controlled dialog, visual interface, keyboard/mouse equivalence, consistency across device classes, and the ability to customize the user interface through user preferences. (For a fuller discussion of these concepts, please refer to CDRL 980, "Programmer's Guide for SUITE".

4.3.3 Metrics for SUITE

SUITE is comprised of about 57 packages totaling to about 14,800 lines of code and 3100 Ada statements. In addition, test code represents about 5000 lines of code or 1700 Ada statements.

SUITE is built on an Ada binding to Motif, which in turn depends on the "C" implementations of Motif and Xlib.

4.4 Comparison of Porting Effort

After some evaluation and discussion with developers, it was decide that a comparison of developer effort in porting applications would not be meaningful. A wide variety of circumstances effect the amount of effort necessary to port software, and in our circumstances it was not possible to hold any of these variables constant. Among the variables effecting porting effort are: Stability of the underlying user interface software, complexity of the user interface concepts implemented by the interface, and experience of the programmer.

To develop (design and implement) the new user interface to APFAT, about 1-2 weeks of effort were required. This time could have been reduced substantially if the SUITE implementation had been complete and stable before work began. There are no available figures for the development of the user interface for APFAT, Version 5, but it was certainly a very short time, since the command line interface style of the interface is extremely simple.

5. IMPLEMENTATION OF SUITE

5.1 Implementation Goals

The Boeing implementation of SUITE was intended as a prototype and only a sample of the possible set of implementations. Since X windows is the most important user interface in the marketplace today, it would be logical for an initial implementation to be on top of X windows.

A key aim of the SUITE implementation was to use an object-oriented methodology. This included the ideals of encapsulation (all data and operations on a given object in a single place), implementation hiding (unnecessary details hidden from the application

programmer), and inheritance (operations and attributes of more general classes (superclasses) of objects being available to more specific subclasses). Furthermore there was a explicit effort made to use features of the Ada language, rather that language extensions or an external library providing general purpose capabilities needed to build components.

Portability is very important to the concept of SUITE. Major goals were to minimize dependencies on the windowing system in the application programmer's interface and, to a lesser extent, on the compilation system in either the application programmer's interface or its implementation.

Another goal of SUITE was extensibility, i.e. the ability for others to define new user interface components for the SUITE object set. The fact that geometric objects (boxes, ovals, etc.) were not included in the current version of SUITE makes this particularly important.

5.2 Analysis of Implementation Options

While Ada supports encapsulation and information hiding, it does not provide any direct support for inheritance. Nevertheless, since the requirements analysis was done on object-oriented lines, it was decided to implement as close as possible to them. The most critical implementation option was thus what method to use for approximating inheritance in Ada.

Six methods for simulating inheritance in Ada were considered. A comparative analysis of them is shown in table 1. The six methods are:

1. *Unisys'.* The method used by Unisys for STARS task UR20, using subtypes and derived types to simulate inheritance, with two packages for each object. Refer to [WAI90], or to the next section.

2. *Unisys' with derived typing.* Similar to (1), but uses derived types and derived operations at points where Unisys uses subtypes. Refer to the next section.

3. *Perez'.* A method developed by Eduardo Perez Perez, which also uses derived types but only one package for each object. Refer to [PER87].

4. *Perez' w/ subtyping and renaming.* Similar to (3), but uses subtyping instead of derived typing and renaming instead of implicitly derived operations.

5. *Hide hierarchy.* Repeating every operation on a superclass in the specifications of its subclasses.

6. *Global data structure.* All attributes for all objects included in a single data structure.

Three criteria are given in the comparison:

1. *Inheritability from Object Programmer Viewpoint.* Does the person coding a SUITE object have to undertake any special actions to simulate inheritance?

2. *Inheritability from Application Programmer Viewpoint.* Does the application programmer have to undertake any special actions to simulate inheritance?

3. *Integrity.* How easily and how badly can data be corrupted or otherwise misused in this method?

As shown in the table, all six of the methods make tradeoffs. Of these methods, it was decided to go with the Unisys method modified to use only derived types (method 2). This achieved a greater distinction between classes and protection against their misuse compared to method (1), at the cost of a greater amount of inconvenience for the application programmer, as discussed in the following section. It achieved a closer simulation of inheritance than the other four methods. The most important reason for eliminating the subtypes was that Unisys was implementing an *Intrinsic* layer -- a set of operations shared by all objects -- while our set of operations was different for each object in order to produce a better "fit" between objects and operations. In Unisys' case, making everything a subtype produced type equivalence, after which it would only be necessary to make the Intrinsic package visible to be able to call any operation. On the other hand, without a shared set of operations the only way to make a superclass operation implicitly available to a subclass was through derived operations, which are generated by type derivation.

An additional method for implementing inheritance in Ada, Classic-Ada, is discussed in section 6.2.

Another major implementation question was the choice of toolkit to use to interface SUITE to X windows. Here the choice was quite limited because few bindings from Ada to X windows libraries were available. The STARS Ada implementation of the Xt Intrinsics by Unisys and SAIC were not available soon enough. A set of Ada bindings to the OSF/Motif toolkit was available for in-house use only from Boeing Commercial Airplanes. Due to availability considerations the Ada bindings to Motif were used.

While STARS software is public domain, the contractors were free to use proprietary or in-house software in their implementations, so that the Motif bindings could be used even though they would not be available to users of SUITE. Since the SUITE implementation was considered a sample and prototype, it was not critical that it maximize its usability by relying only on public domain software.

TABLE 1 Comparison of Methods for Simulating Inheritance

Method	Inheritability from object programmer viewpoint	Inheritability from application programmer viewpoint	Integrity
Unisys'	Full	Full	No protection against violations of class hierarchy -- consequences could be disastrous
Unisys' (w/ derived typing instead of subtyping)	Must explicitly provide operations that are not implicitly derived	Some checked type conversions necessary	Allows violations of class hierarchy if checked type conversions are done
Perez'	Must explicitly provide operations that are not implicitly derived	Must explicitly convert objects to superclass type(s) or use **link** operation to link instances of class & superclasses	Full
Perez' w/ subtyping & renaming	Must explicitly convert objects to superclass type(s); must provide renaming clauses for all inherited operations & attributes	Must use **link** operation to link instances of class & superclasses when updating objects	Full
Hide hierarchy	Must provide subprogram bodies for all operations	Full	Many subprogram calls requires pragma INLINE for efficiency

TABLE 1 Comparison of Methods for Simulating Inheritance [continued]

Method	Inheritability from object programmer viewpoint	Inheritability from application programmer viewpoint	Integrity
Global data structure	Full	Full	No protection against violations of class hierarchy (although data is not destroyed); must recompile entire system if any attribute is added/changed; meaningless data is visible to operations (stamp coupling)

5.3 Description of Implementation

5.3.1 The Class Hierarchy

As already mentioned, SUITE attempts to simulate inheritance by a method similar to the one adopted by Unisys in task UR20. This method involves two sets of packages for each object.

The first package is the "public" spec; this is the interface intended for use by the application programmer. Each class in the public spec is represented by a derived type, which in Ada allows it to inherit the operations defined for its parent type, including the operations that the parent itself inherited. (Alas, constants and exceptions defined for the class are not inherited, nor are operations in which an object of the parent type is not a parameter, although there were only a few of the latter).

The second package is the "private" spec. This includes information that other SUITE classes need but the application programmer does not. Since other classes (packages) must use the data, it cannot be hidden in the body of the public package. The primary information in this package is the record used to contain the attribute data for the class. As shown in figure 1, each class record is divided into two parts, an "inherited" part and an "added" part. The inherited part is the superclass' data fields, so that each class has all of the attributes of its superclasses. The added part contains the information pertinent only for the new class and its subclasses.* To get an attribute of

*In BR22, SAIC's class record repeats the attributes of its superclass rather than encapsulating them as an "inherited part". This permits easier access to the superclass attributes at the price of redundancy.

some distant superclass, one travels through the "inherited" parts up the superclass tree until the proper "added part" becomes accessible. Unfortunately, the statements that traverse up the inheritance tree do not indicate which class contains the accessed attribute, since the access appears as:

<object>.Inherited_Part.Inherited_Part...Inherited_Part.Added_Part

The top of the class hierarchy, the class that is the superclass of all other classes, is called *Core*. For the Core class only, the data type in the public spec is a pointer to the data type in the private spec (the Core data record). Since all other public spec types are derived from the public Core_Type, that means that all of the public types are also pointers to the Core data. In order to access the data for their own classes, an unchecked conversion must be performed which converts from a pointer at one object (the Core record) to a pointer at another object (the class' own data record).

The mapping of class data into memory is strictly controlled. The inherited part is always placed in memory before the added part. The inherited part may, in turn, have its own inherited and added parts, which must also obey this rule. This structure continues up to the innermost class (Core) which as a result is always is placed at the very top of a class' data structure. If the added part contains components, then those components will have their own inherited and added parts; care must be taken to avoid confusing between the core part of an object and the core parts of its components.

The purpose of the strict control over the data structure is to ensure that inheritance will work. An operation on a class takes an object of a "public" data type, which as already mentioned is either Core or a derivation of it. If the attribute referenced is part of Core, access to the private type is simple. If it is in some subclass, then unchecked type conversions must be performed to convert the pointer to the private Core into a pointer to the private version of whatever class contains the attribute. Without the strict control, there would be no guarantee that the proper data would be accessible after the type conversion. Take figure 1 as an example. An Application Panel is derived from a Resourced Object, which in turn is derived from Core. An operation on an Application Panel initially receives a pointer to Core and converts it -- unchecked -- into a pointer to Application Panel: from the point of view of a pointer to a Core object, the attributes of an Application Panel are nonexistent; after the conversion, the Application Panel fields are made visible. But only through the strict control is it assured that the application panel fields are actually represented in memory by application panel data; otherwise a compiler could rearrange the Core and Application Panel fields in memory and create havoc. Without the representation specifications, the code would be considered erroneous by Ada's standards.

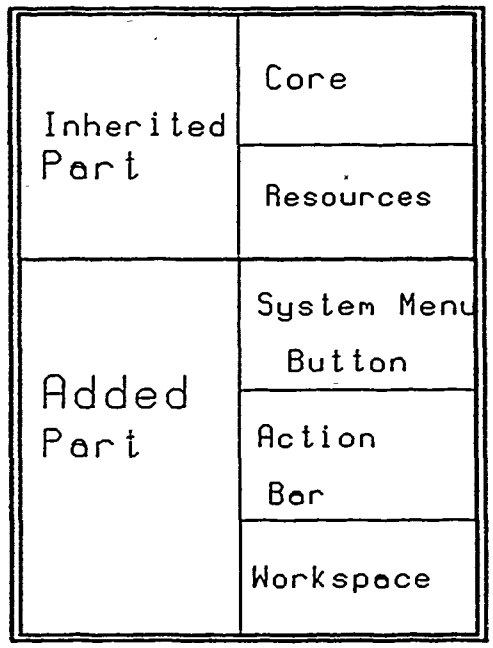

Figure 1 Composition of a SUITE Component (Application Panel)

The mapping of inheritance to Ada has some limitations. In particular, there are a number of operations in which one object of class X is inserted to or retrieved from another object of class Y. All of these operations are defined in the package corresponding to Y. Now suppose we have subclasses for X and Y, called X' and Y', respectively and an operation *f*(X, Y), which as mentioned above will be placed in Y's package. Ada's type derivation mechanism will create a derived operation *f*(X, Y'). not *f*(X', Y') which is what we really need. Placing the operation in X's package will not help, because then the derivation produces the operation *f*(X', Y), which is also wrong. In this case, the application programmer must perform an explicit (checked) type conversion. For example, consider the classes *Item* and *List* and the operation

```
Add (An_Item: in Item; , To_List: in out List);
```

that is defined in List's package. If we have a subclass for Item called *Button* and a subclass for list called *Menu*, then we will need an operation

```
Add (An_Item: in Button; To_List: in out Menu);
```

but the derived operation will actually have the parameter profile

```
Add (An_Item: in Item; To_List: in out Menu);
```

To invoke the derived operation, the programmer must convert his button from type

Button to type Item, using a single checked type conversion.

All components and some attributes of each object are defined internally as part of the object. Other attributes, however, are defined as *resources*. A resource, unlike an internally-defined attribute, can be set by the user with a file by stating the name of the object (or its class to affect all objects in the class), the resource name, and the desired value. In general, resources affect only the view of an object and not its behavior. The decision of which attributes should be resources and which should not were a matter of balancing user tailorability against the ability of SUITE to control the attribute to ensure proper behavior.

The application programmer can modify and query internally-defined attributes with operations defined in the package of the class containing the attribute. The operations on resources, on the other hand, are defined by a single class (package), *Resourced_Object*. It provides a common set of operations over all resources using a set of generics that other SUITE objects instantiate to define what resources they will provide. Therefore, while the application programmer needs to refer to the Resourced_Object package to know the set of available operations and their parameter profiles, he would not actually be using it directly (except for the resources defined for all classes) but rather the generic instantiations in the other SUITE objects.

The Ada code for a sample SUITE object is shown in appendix A.

5.3.2 The Windowing System Interface

The sample SUITE implementation is on top of X windows, and specifically the X toolkit intrinsics (Xt), the OSF/Motif widget set, and the Boeing Commercial Airplane's Ada binding to Motif.

In several respects, Xt was inimical to our object-oriented design and to object-orientedness in Ada.

We followed the philosophy advocated by Grady Booch [BOO87] and Ed Berard [BER89] that objects be ignorant of the context in which they are used; for example, an object would have no knowledge of whether it would be placed in a set, tree, table, etc. and would not reference any of those complex data structures. Xt, however, *forces* contextual knowledge on its users. This is due to the "parent" parameter required when creating an Xt object (widget), which specifies what object the new widget is to be inserted into (its "parent"). Furthermore, widgets must be created in a top-down order in Xt, i.e. a parent widget must be created before any child widgets can be created. SUITE has no such restriction on the order of creation of its objects. Furthermore, the mapping of some SUITE objects to Motif widgets was context-dependent, i.e. depended on what type of object they were placed into. For example, a SUITE selection list could be implemented as a Motif menu bar (if placed in an action bar), a

pulldown menu (if attached to a single menu choice within an action bar), or a popup menu (if attached to the background). If this contextual information was hard-coded into the parents, then extensibility would be reduced. Resolving our methodology with Xt's in order to successfully create widgets proved to be one of the most difficult, if not the most difficult, issue during implementation. After considering a number of methods which failed or were foreseen to fail, we adopted a method in which each class would have its own widget creation procedure. If the parent widget were already created, this procedure would be invoked immediately. Otherwise, it would be invoked later, when the object was inserted into a given parent. This method involves procedure variables, another feature not readily available in Ada. Unisys' UR20 report discusses a complicated method of providing procedure variables, which time did not permit us to implement. Therefore, we adopted a temporary restriction that all objects could only be inserted in a top-down order, i.e. no object could be inserted into a second object that was not itself inserted into something else (except for objects of the topmost class of the parenting hierarchy, Application.) An intermediate-term solution would be to hard-code the entire widget creation mechanism within the application, which would allow objects to be inserted in any order but would reduce extensibility.

Xt allows an application to respond to user action (e.g. selecting a push button by pressing it) by invoking a given procedure each time the user performs a given action. These procedures are referred to as *callbacks*. The Xt programmer can specify the callback procedure corresponding to each object that the user can select. However, this is another case requiring procedure variables. If the callback is parameterless, it can be passed as a variable by its address. If it has parameters, some additional means are necessary to provide actual values for those parameters. This is an even more complicated procedure variable problem than the one needed to create a widget, since the widget creation procedures would all have the same number and type of parameters while callback procedures could have any number of parameters of any type. Unisys does present an even more complicated method of simulating procedure variables for this case, which involves no less than passing around the entire calling environment. Rather than use up the entire project time to implement it, we adopted a method using a set of generic packages with subprogram parameters, which could accommodate callbacks with up to three parameters. (If the application programmer had a callback with more than three parameters, he would have to bundle some of them into a record and unbundle them when his callback was invoked.)

The callback procedures were another case in which the class hierarchy partially failed, since procedures in nested generic packages are not derivable in Ada. When setting a callback procedure for a subclass object, a type conversion again was necessary. And if one of the parameters to the callback procedure was also a generic parameter, then even type conversion would be insufficient, as shown in the following example:

```
type Base is ...
```

```
type Derivation is new Base;
B: Base;
D: Derivation;
generic
    type T is limited private;
    with procedure Callback (Using: in T);
package Operations is
    procedure Do_It (To: in T);
```

Suppose you want to call Do_It on both B and D. You must instantiate Operations twice, once for each type, even though one is a derivation of the other. This is caused by the LRM 6.4.1, which states that if a type conversion is used as the actual parameter for Do_It, then the "type mark" of the conversion must match the type mark of the formal parameter, i.e. the conversion must be from Derivation to T, which would be illegal because T is a generic type.

Some other Xt idiosyncrasies created problems to try to hide their effects from the application programmer. One was that no commands to display or undisplay windows would take effect on the screen until "buffer flushing" was done, by calling either XtPending or XtNextEvent. In most cases, the buffer flushing could be hidden away from the application programmer during displaying or undisplaying commands, but there still may be some circumstances where the application programmer would have to do a "flush" himself. (We accomplished the hiding at the very end of the project and have not determined if we covered all cases.)

5.4 Conclusions

The application programmer's interface (API) for SUITE appears to have succeeded in its goal of hiding the underlying windowing system (X windows) in its application programmer's interface (public specs). While some Xt concepts have been retained in the API (e.g. resources), steps have been taken to allow such concepts to be ported to other windowing systems. (With resources, a complete interface has been designed for resource management that hides the Xt resource manager from the application programmer.)

We have also been successful in maximizing extensibility, at least in concept. It is possible that shorter-term implementations would have some hard-coding that would hinder extensibility.

SUITE was less successful in minimizing dependencies on the Ada compilation system. The use of representation specs is not a problem -- they actually enhance portability, for the reasons given in section 5.3.1. The main culprit was the liberal use of System.Address and the 'Address attribute in implementing procedure variables. This would produce big headaches in trying to port SUITE to the VAX

Ada compiler, which does not support these constructs.

SUITE has been partially succ ful in implementing a class hierarchy, using derived operations. We were una e to include attributes or exceptions in the class hierarchy, and the previous section described several cases in which the application programmer would have to perform checked type conversion. However, we do not know of any method without the use of additional tools (see section 6.2) that would more closely simulate inheritance in Ada without permitting the class hierarchy to be violated more easily.

6. RECOMMENDATIONS FOR FUTURE WORK

6.1 Refine Foley's Concept of "Selection Interaction Task"

One of the more difficult and reoccurring problems of this task was to arrive at a satisfactory abstraction for Foley's concept of selection interaction task [Foley 84], [CDRL980]. This concept guided the definition of the following suite components: Selection item, selection list, command item, command list, toggle item, menu command item, multi- and single-choice lists, and text selection. This part of the SUITE class hierarchy was very unstable, possibly reflecting the fact that the correct abstraction was not being used.

Late in the task we realized that there might be a substantial difference between the selection of "actions" as opposed to the selection of "objects." Components adapted to "action" selection include command item, command list, toggle items, and single and multi-choice lists. Components adapted to "object" selection include text selection.

Additional analysis of the subject might lead to a redefinition of classes and operations adapted to each type of selection. A subsequent re-organization of the SUITE class hierarchy would then be required.

6.2 Explore Alternative Implementations that use Language Extensions to Ada

Classic-Ada, from Software Productivity Solutions (SPS), is an Ada preprocessor intended to fully implement a class hierarchy according to the precepts of object-orientedness, including full inheritance.

The use of Classic-Ada somewhat conflicted with our goal of not using language extensions. However, later in the task, when implementation difficulties were apparent, we decided that it was appropriate to re-evaluate the potential benefits of using language extensions. We received a temporary evaluation copy of it near the end of the R24 schedule. While time has not permitted a sufficient re-implementation of SUITE to completely compare it to our Ada implementation, we have done enough to indicate that Classic-Ada might be enormously beneficial. Its greatest impact in SUITE was for those classes that heavily used operations to insert SUITE objects into other SUITE objects, which required a large amount of unchecked conversions in the standard Ada implementation but only a single line of Classic-Ada code. One such class, Application Panel, required only one-sixth as much code in Classic-Ada as conventional Ada.

A critical question in the ability to implement SUITE in Classic-Ada is callback procedures (procedure variables). SPS claimed that these could be implemented

by making the callback procedure itself an object; we did not get the opportunity to put this claim to the test.

The suggested work in this area would be to:

1. further implement the Classic-Ada version of SUITE enough at least to produce some graphic output, which at this point mainly requires calls to create Motif widgets.

2. attempt to implement callback procedures in Classic-Ada.

6.3 Demonstrate Application Portability

We would prefer to demonstrate, rather than to just assert, that SUITE enhances the portability of applications across a variety of underlying user interface software and classes of display devices. Currently our assertion of application portability is based only on the abstract nature of the SUITE user interface concepts and the corresponding application programmer's interface. A working prototype would certainly be convincing everyone involved, including ourselves! A very convincing demonstration could be made by porting SUITE to a character-oriented display device, such as an ANSI terminal. It is our belief that the SUITE interface could be adapted to such a device without significant violence being done to the original concept. Such a demonstration would have to be evaluated against the current goals and objectives of the STARS program, which would appear to give less emphasis to application portability across a very wide range of classes of display device and user interface technology.

6.4 Extend SUITE to Application-specific Components

In the mission statement for BR24 [CDRL 980] it was stated: "SUITE is intended to support a variety of applications in the domain of systems and software engineering. The current scope of SUITE is limited to user interface (UI) components that are useful across the entire range of these applications. Components that are application-domain specific are beyond scope, but could be considered at a later date, as consistent practice evolves in the field and when additional resources and expertise are available."

There is a need for a wide variety of user interface components to support applications development in the software engineering domain: Graph components like arc and nodes with subclasses implementing the semantics of different types of graphs; presentation graphics components for representing tabular data, such as plots, charts, and tables.

6.5 Clean–up SUITE Implementation

The SUITE implementation would benefit from a review and consolidation of code. The following paragraphs refer to detailed modifications to the SUITE implementation that would improve robustness, increase consistency, or simplify usage. The full understanding of this list requires familiarity with the Ada implementation of SUITE.

Add In_Enabled_State as a parameter to Selection_Item.Create; Alternatively, specify that all newly created object are by default in the enabled state.

Evaluate whether it would be beneficial to implement menus using SPC menus. The operation "set_callback.." in menu_command_item subclass must be redefined to take a selection list (specifically) as an argument. The new body would then build a menu tree from scratch.

Change Default_Value in package ID to Nul and add an exception ID_Is_Nul.

Each private spec should have a Create_Widget (Resourced_Object_Type) procedure which will be invoked using a system–independent procedure variable scheme in the long run and by hard–coding into the global widget creation procedure in the short run.

Destroy the old widget each time you add an object to another.

Replace use of "Object" in operations with more definitive terms.

Add a note that all objects are initially in uncreated state.

Each call to set a resource value will have to produce a call to Xt_Realize_Widget due to the behavior of Xt.

Add an operation in selection list to add an object to the end of the list?

Un–comment out the record representation clauses –– at the last minute!

Replace calls of Set_The_Class_Of (and Set_The_ID_Of?) in Create with direct record references.

Modify Application_Panel.Destroy (and others?) to only delete a widget if it is not Null_Widget.

In Menu_Command_Item, remove the conversions to Resourced_Object.

Ensure that all classes are one linked word, with a capital as the first letter only.

Create Interactor and Panel subclasses. The Interactor subclass will have no children.

Rename package Widget_List to Resourced_Object_List.

Have a cleaner way of stopping Xt. See p. 36 of Xt Intrinsics Manual; should call XtDestroy application context and unix exit. This code should probably be put in Application.Destroy.

Inserting items into a list doesn't give the expected order. This is because the default insert procedure for composite widgets is to insert them a the head of the list. The behavior of the widget should be modified by replacing the Motif insert callback with one that inserts according to position.

Parameters that are ID or Class need only be of type String, not A_String, since a String parameter accepts strings of any length. (The SUITE bodies can do the conversion.)

Some classes of objects (e.g. selection lists) can validly either exist independently (for selection lists, as a popup menu) or as a component of another object (for selection lists, as a component of a dialog). In the first case, the object would be a secondary panel; in the second case, it would not. Whether it is a secondary panel or not would not be known until the object was inserted into something else, which means that if Secondary Panel is a class, its class would not be known at creation time. Therefore, "Secondary Panel" (independent displayability) should probably really be an attribute rather than a class.

Is_Empty should probably be removed from Selection_List, since Nul and The_Size_Of handle it.

Routines to get/set resources of different types could be rewritten as generics. (Widget_Resources, Boolean_Resources and others).

Increase length of Class and ID types to 50.

6.6 Port the SUITE Implementation to the STARS X/Ada Interface

Due to availability, SUITE was not developed on top of the STARS X/Ada interface. This would be a natural and desirable transition when a complete STARS X/Ada Toolkit (Intrinsics and Widget Set) become available.

A. SAMPLE CODE FOR A SUITE CLASS

A.1 Spec

```
with Component;
with ID;
package Sample is

  type Sample_Type is new Component.Component_Type;
    -- This particular object is defined as a subclass of Component class.

  procedure Create
    (A_Sample: out Sample_Type;
     With_ID: in ID.ID_Type := ID.Default_Value;
     Activated: in Boolean := True);
--| Exceptions_Raised>
--|   Out_Of_Memory
--|| Exceptions_Raised>

  procedure Destroy
    (The_Sample: in out Sample_Type);
--| Exceptions_Raised>
--|   Object_Destroyed
--|| Exceptions_Raised>


  -- Operations on the sample class' "activity" state
  procedure Activate
    (The_Sample: in out Sample_Type);
--| Exceptions_Raised>
--|   Object_Destroyed
--|| Exceptions_Raised>

  procedure Deactivate
    (The_Sample: in out Sample_Type);
--| Exceptions_Raised>
--|   Object_Destroyed
--|| Exceptions_Raised>

  function Is_Activated
    (The_Sample: Sample_Type) return Boolean;
--| Exceptions_Raised>
--|   Object_Destroyed
--|| Exceptions_Raised>
```

```
  -- Operations to add/change/retrieve a component that is nested
  -- inside the sample object.
  procedure Set_The_Inner_Window_Of
    (The_Sample: in out Sample_Type;
     To: in Component.Component_Type);
    -- Any object that is a subclass of Component can be an inner window,
    -- according to this definition.

  function The_Inner_Window_Of
    (The_Sample: Sample_Type) return Component.Component_Type;

  Object_Not_Created: exception;
  Out_Of_Memory: exception;

end Sample;
```

The Sample class defined in this package is a subclass of the Component class. The type derivation statement will derive all operations in the Component package on Component_Type types, as well as those that are inherited by Component_Type types. (Actually, there are no operations defined in the Component package, but it does inherit operations from the Core and Resourced_Object packages. Most SUITE operations come in sets: one or more operations to change a specific state/property (constructors) together with an operation to query the current state value (selector). In this case, there is an "activation" state with two constructors for it (Activate and Deactivate) together with one Selector (Is_Activated). There is a pair of operations to change the sample object's inner window. The "inner window" property is itself a SUITE component, the "activation" property is not another component but rather an *attribute*. In the SUITE terminology, Existence is itself a state, which is changed by Create and Destroy. (In this case, there is no query operation.) SUITE objects are all derived from an access type (Core_Type), so all objects are initially in a Destroyed state, that is null. (It may be curious to think of an object to be destroyed before it has ever been created, but from the point of view of the SUITE programmer, there is really no difference between a Destroyed object and an uninitialized one.) Passing an Destroyed object to any operation except Create raises the exception Object_Not_Created. Note that a object passed to Destroy can later be re-Created.

A.2 Private Spec

```
with Class;
with Component;
with Resourced_Object_Private;
package Sample_Private is

  type Added_Information_Type is record
```

```
    Active: Boolean;
    Its_Inner_Window: Component.Component_Type;
  end record;
    -- The type of a component here is its PUBLIC type.
  Added_Information_Default_Value: constant Added_Information_Type
    := (Active => True,
        Its_Inner_Window => null
       );

  type Sample_Record_Type is record
    Inherited_Part: Resourced_Object_Private.Resourced_Object_Record_Type;
    Added_Part: Added_Information_Type;
  end record;

  -- This rep clause is to enforce the order of the record components.
  Added_Information_Size: constant := 48;
  for Sample_Record_Type use record
    Inherited_Part at 0 range
      0..Resourced_Object_Private.Resourced_Object_Type_Size-1;
    Added_Part at Resourced_Object_Private.Resourced_Object_Type_Size
      range 0..Added_Information_Size-1;
  end record;

  Default_Value: constant Sample_Record_Type
    := (Inherited_Part => Resourced_Object_Private.Default_Value,
        Added_Part => Added_Information_Default_Value);

  Sample_Record_Size: constant := 24104;
--| Description>
--| The number of bits needed for an object of type Sample_Record_Type.
--| It is a shame to have to hard-wire the size like this rather than rely on the
--| 'Size attribute.
--| However, 'Size is not legal in record representation clauses.
--|| Description>


  Sample_Class: constant Class.Class_Type
    := Class.The_Class_Version_Of ("Sample");
--| Description>
--|   This is used by the resource manager to determine the set of resources
--|   available for a Sample object.
--|| Description>

procedure Create_The_Widget_For
  (Object: in out Sample_Type);
```

```
end Sample_Private;
```

The "private spec" is where the types for actually storing the class data are defined. In this case, Sample_Record_Type holds the data for Sample_Type. The application programmer never needs to declare anything as a Sample_Record_Type -- but other SUITE classes do. The result is that this type declaration must be in a spec -- but it is better to have it in a different spec from the one used by the application programmer.

Note that the data field in Sample_Record_Type to hold the Inner_Window component is of the public (Access) rather than the private (Record) type. The reason is that when the body of Sample references the Inner_Window component of a Sample object, it might wish to invoke an operation on Inner_Window, which takes the public type as parameter.

As already mentioned, each private spec includes an inherited part and an added part. The inherited part, which represents the superclass information, is normally the record associated with its superclass, which in the case of Sample_Type is Component. Since Component has no data of its own, the record of Component's superclass, that is Resourced_Object, is used.

Each private spec should define a default value for its class. This value is not used by the body of Sample, but it is used by any class that is a subclass of Sample (or will be later on -- extensibility!) in the same way that Sample uses the default value of its superclass,

A.3 Body

```
with Class;
with Sample_Private;
with Unchecked_Conversion;
with Unchecked_Deallocation;
with X_Toolkit_Intrinsics_OSF;
with Xm_Widget_Set;
package body Sample is

  subtype Sample_Record_Type is
    Sample_Private.Sample_Record_Type;
  type Sample_Record_Access_Type is access
    Sample_Record_Type;

  -- These type conversions allow access to the Sample_Record_Type
  -- data fields.
  function The_Sample_Record_Access_Version_Of is new
    Unchecked_Conversion
```

```
  (Source => Sample_Type,
   Target => Sample_Record_Access_Type
  );
function The_Sample_Version_Of is new Unchecked_Conversion
  (Source => Sample_Record_Access_Type,
   Target => Sample_Type
  );


procedure Create
  (A_Sample: out Sample_Type;
   With_ID: in ID.ID_Type := ID.Default_Value;
   Activated: in Boolean := True) is

  Motif_Sample_Class:
    constant X_Toolkit_Intrinsics_OSF.Widget_Class
      := Xm_Widget_Set.Xm_Row_Column_Widget_Class;

  -- Declarations used to access the Sample record fields
  -- and convert it to a Sample_Type.
 The_Sample_Record_Access:
   Sample_Record_Access_Type;


  Its_ID: ID.ID_Type renames With_ID;

begin
 The_Sample_Record_Access := new
    Sample_Record_Type'(Sample_Private.Default_Value);

  -- Set the values for the "Resourced_The_Sample" part of Object.
 The_Sample_Record_Access.Inherited_Part.Added_Part
    .Its_Widget_Class
      := Motif_Sample_Class;

  The_Sample := The_Sample_Version_Of (The_Sample_Record_Access);
    -- Initializing The_Sample directly would have only initialized enough
    -- space for the Core part, because Sample_Type is a derivation
    -- of a pointer to Core.

  -- Set the values for the "core" part of The_Sample.
  The_Sample.Its_ID := Its_ID;
  The_Sample.Its_Class := Sample_Private.Sample_Class;

exception
```

```
  when Storage_Error =>
    raise Out_Of_Memory;
end Create;



procedure Destroy
  (The_Sample: in out Sample_Type) is

  procedure Do_Destruction_Of is new Unchecked_Deallocation
    (Object => Sample_Record_Type,
     Name => Sample_Record_Access_Type);

  The_Sample_Record_Access:
    Sample_Record_Access_Type :=
      The_Sample_Record_Access_Version_Of (The_Sample);

begin
  If The_Sample = Null then
    raise Object_Not_Created;
  else
    Do_Destruction_Of (The_Sample_Record_Access);
    The_Sample := null;
  end if;
end Destroy;



-- Operations on the sample class' "activity" state
procedure Activate
  (The_Sample: in out Sample_Type) is

  The_Sample_Record_Access:
    Sample_Record_Access_Type :=
      The_Sample_Record_Access_Version_Of (The_Sample);

begin
  If The_Sample = Null then
    raise Object_Not_Created;
  else
    The_Sample_Record_Access.Added_Part.Active := True;
  end if;
end Activate;
```

```
  procedure Deactivate
    (The_Sample: in out Sample_Type) is separate;

  function Is_Activated
    (The_Sample: Sample_Type) return Boolean is separate;

  function The_Inner_Window_Of
    (The_Sample: Sample_Type) return Component.Component_Type is separate;

  procedure Set_The_Inner_Window_Of
    (The_Sample: in out Sample_Type;
     To: in Component.Component_Type) is separate;

end Sample;
```

All operations in SUITE packages receive as a parameter an object of Core_Type or a derivation of Core_Type. Core_Type is a pointer to the Core attributes. In order to get to any other attributes, including Sample's own attributes, unchecked type conversion must be performed. In the body of Sample, this is done by invoking function The_Sample_Record_Access_Version_Of. Likewise, a call to this function must be performed during Create so that enough space is allocated for an entire Sample record, not just the Core attributes.*

All visible operations other than Create should first check to ensure that its object parameter is non-null.

An operation that inserts a SUITE component into another SUITE component (e.g. Set_The_Inner_Window_Of) is the most difficult case to implement in Motif/Xt, so it is presented and discussed separately from the rest of the body.

A.3.1 A Component Insertion Operation

```
with Core;
with Resourced_Object;
with Resourced_Object_Private;
with Widget_List;
separate (Sample)
  procedure Set_The_Inner_Window_Of
    (The_Sample: in out Sample_Type;
     To: in Component.Component_Type) is

    subtype Inner_Window_Type is Component.Component_Type;
```

* Because a pointer to a Core object must be returned from the operation, the pointer to the Sample record must be re-converted to a pointer to Core before exiting the operation.

```
The_Inner_Window: Inner_Window_Type renames To;

subtype Resourced_Object_Type is
  Resourced_Object.Resourced_Object_Type;


-- Type conversions to allow access to Inner_ Vindow record fields.
type Resourced_Object_Record_Access_Type is
  access Resourced_Object_Private.Resourced_Object_Record_Type;
function The_Resourced_Object_Version_Of is new Unchecked_Conversion
  (Source => Component.Component_Type,
   Target => Resourced_Object_Record_Access_Type
  );
The_Inner_Window_Record_Access:
  Resourced_Object_Record_Access_Type
    := The_Resourced_Object_Version_Of (The_Inner_Window);

-- Type conversions to allow access to the Core subclass components
-- of The_Sample
The_Sample_Record_Access:
  Sample_Record_Access_Type
    := The_Sample_Record_Access_Version_Of (The_Sample);


-- There are two Inner_Window records referred to in this subroutine.
-- The first is the "workspace" component of The_Sample, and the second
-- is the Inner_Window parameter.  This is the former of the two.
The_Objects_Inner_Window_Field: Inner_Window_Type
  renames The_Sample_Record_Access.Added_Part.Its_Inner_Window;
The_Inner_Window_Field_Record_Access:
  Resourced_Object_Record_Access_Type;


function "=" (x, y: X_Toolkit_Intrinsics_OSF.Widget) return Boolean
  renames X_Toolkit_Intrinsics_OSF."=";

begin
  if The_Sample = Null then
    raise Object_Not_Created;
  else

    The_Sample_Record_Access.Added_Part.Its_Inner_Window
      := The_Inner_Window;
    The_Inner_Window_Field_Record_Access
      := The_Resourced_Object_Version_Of
```

```
        (The_Sample_Record_Access.Added_Part.Its_Inner_Window);

      -- This part deals with creation of the widget for The_Inner_Window.
      if The_Sample_Record_Access.Inherited_Part.Added_Part.Its_Resources
        = X_Toolkit_Intrinsics_OSF.Null_Widget then
        -- The_Sample's widget hasn't been created yet
        -- Save parent-child information
--                                    if    The_Sample.Its_Widget_Tree    =
Unordered_Unbounded_Managed_Tree.Empty then
--          -- Create a new tree with application panel as its root and
--          -- the workspace's tree as its subtree
--          Widget_Tree.Operations.Add_Root
--            (To_Make => The_Sample.Its_Widget_Tree,
--             From_Item => The_Sample,
--             And_Subtree => The_Inner_Window_Record.Widget_Tree
--            );

--        else -- The_Sample's widget tree already exists;
--               -- Add the workspace as a new child
          Widget_List.Add_Object
                                                          (To_List     =>
The_Sample_Record_Access.Inherited_Part.Added_Part.Its_Children,
           With_Value => Resourced_Object_Type(The_Inner_Window)
          );
         The_Inner_Window_Field_Record_Access.Added_Part.Its_Parent :=
           Resourced_Object_Type(Core.Core_Type(The_Sample));
            -- The_Sample is of Sample type;
            -- we need a Resourced_Object type.
--        end if;

      else -- The_Sample's widget exists; Go ahead and create the Inner_Window
widget

        -- In the short term, this actually creates the widget
        -- In the medium term, this will do nothing and the widget will
        -- be created elsewhere.
        -- In the long term, this will invoke a subroutine to create the
        -- widget.
        The_Inner_Window_Field_Record_Access.Added_Part.Its_Resources
          := X_Toolkit_Intrinsics_OSF.Xt_Create_Managed_Widget
           (Name => ID.The_String_Version_Of (The_Inner_Window.Its_ID),
                                                          Class        =>
The_Inner_Window_Record_Access.Added_Part.Its_Widget_Class,
                                                          Parent       =>
The_Sample_Record_Access.Inherited_Part.Added_Part.Its_Resources
```

```
        );

    end if;
  end if;
end Set_The_Inner_Window_Of;
```

Until procedure variables are supported, this procedure just invokes an operation to create the inner component's Motif widget. This will usually be through a call of Xt_Create_Managed_Widget, unless Motif provides a convenience routine for the desired widget class.

A.4 Private Body

```
with Sample;
with Resourced_Object;
with Unchecked_Conversion;
package body Sample_Private is

  procedure Create_The_Widget_For
    (Object: in out Sample.Sample_Type) is

    -- These declarations allow access to the Sample data
    -- fields of Object.
    subtype Sample_Type is Sample.Sample_Type;
    type Sample_Record_Access_Type is access Sample_Record_Type;
    function The_Sample_Record_Access_Version_Of is new
      Unchecked_Conversion
        (Source => Sample_Type,
         Target => Sample_Record_Access_Type);
    The_Sample_Record_Access:
      Sample_Record_Access_Type
        := The_Sample_Record_Access_Version_Of (Object);

    -- Objects needed to create the top-level (application) shell
    Some_Class: constant String := "Sample_class";

    -- These declarations provide direct access to Resourced_Object fields.
    subtype Resourced_Object_Type is Resourced_Object.Resourced_Object_Type;
    type Resourced_Object_Record_Access_Type is access
      Resourced_Object_Private.Resourced_Object_Record_Type;
    function The_Resourced_Record_Access_Version_Of is new
      Unchecked_Conversion
        (Source => Sample_Type,
         Target => Resourced_Object_Record_Access_Type);
```

```
    The_Resourced_Version_Of_Object: Resourced_Object_Type :=
      Resourced_Object_Type (Object);
        -- Holds value of (checked) type conversion from File_Selection_Type.
    The_Sample_Resources:
      Resourced_Object.Resources_Type;
        -- Holds the Xt resources (widget) data for Object.

      -- The object that contains Object.
    The_Parent: Resourced_Object_Type
      := Resourced_Object_Type(The_Parent_Of (Object));
        -- The_Parent_Of inherited from Resourced_Object.

  begin  -- (Create_The_Widget_For)
    -- Create the Sample's widget.
    The_Sample_Resources :=
        X_Toolkit_Intrinsics_OSF.Xt_Create_Managed_Widget
          (Name => ID.The_String_Version_Of
                    (Object.Its_ID),
           Class =>Sample.The_Widget_Class_Of (Object),
           Parent => Resourced_Object.The_Resources_Of (The_Parent)
          );
   Set_The_Resources_Of
     (Object, To => The_Sample_Resources);
        -- Operation inherited from Resourced_Object
  end Create_The_Widget_For;

end Sample_Private;
```

The *Boeing* Company

ACTIVE SHEET RECORD

SHEET NO.	REVLTR	ADDED SHEETS				SHEET NO.	REVLTR	ADDED SHEETS			
		SHEET NO.	REVLTR	SHEET NO.	REVLTR			SHEET NO.	REVLTR	SHEET NO.	REVLTR
1						28					
2						29					
3						30					
4						31					
5						32					
6						33					
7						34					
8						35					
9						36					
10						37					
11						38					
12						39					
13						40					
14						41					
15						42					
16						43					
17						44					
18											
19											
20											
21											
22											
23											
24											
25											
26											
27											

THE *BOEING* COMPANY

REVISIONS			
LTR	DESCRIPTION	DATE	APPROVAL